A Data Structure for Circular String Analysis and Visualization *

Dinesh P. Mehta †‡ Sartaj Sahni†

Technical Report 25

Abstract

Circular strings are used to represent circular genomes in molecular biology, polygons in computer graphics and computational geometry, and closed curves in computer vision. In this paper we extend techniques which have so far been successfully applied to the analysis and visualization of linear strings to circular strings by defining a data structure for circular strings. Efficient (often optimal) algorithms that support these techniques are presented.

Keywords and Phrases:

Circular strings, visualization, analysis, directed acyclic word graphs.

*This research was supported in part by the National Science Foundation under grant MIP 86-17374.

†Dept. of Computer and Information Sciences, University of Florida, Gainesville, FL 32611

‡Dept. of Computer Science, University of Minnesota, Minneapolis, MN 55455

1 Introduction

The circular string data type is used to represent a number of objects such as circular genomes, polygons, and closed curves. Research in molecular biology involves the identification of recurring patterns in data and hypothesizing about their causes and/or effects [1, 2]. Research in pattern recognition and computer vision involves detecting similarities within an object or between objects [3].

Detecting patterns visually is tedious and prone to error. In [4], a model was proposed to alleviate this problem. The model consists of identifying all recurring patterns in a string and highlighting identical patterns in the same color.

[4] also listed a number of queries that the model would support. In [5], efficient (mostly optimal) algorithms were proposed for some of these queries for linear strings. These algorithms perform operations and traversals on the symmetric compact directed acyclic word graph (scdawg) [6] of the linear string. The scdawg, which is used to represent a string or a set of strings, evolved from other string data structures such as position trees, suffix trees, directed acyclic word graphs, etc [7, 8, 9, 10].

One approach for extending these techniques to circular strings is to arbitrarily break the circular string at some point so that it becomes a linear string. Techniques for linear strings may then be applied to it. However, this has the disadvantage that some significant patterns in the circular string may be lost because the patterns were broken when linearizing the string. Indeed, this would defeat the purpose of representing objects by circular strings.

[3] defined a polygon structure graph, which is an extension of suffix trees to circular strings. However, the suffix tree is not as powerful as the scdawg and cannot be used to solve some of the problems that the scdawg can solve. In this paper, we define an scdawg for circular strings. Algorithms in [5] and [6] which make use of the scdawg for linear strings can then be extended to circular strings with minor modifications. The extended algorithms continue to have the same efficient time and space complexities. Further, the extensions take the form of postprocessing or preprocessing steps which are simple to add on to a system built for linear strings, particularly in an object oriented language.

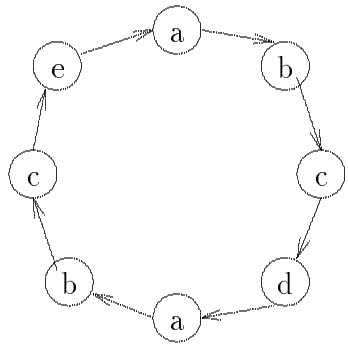


Figure 1: Circular string

Section 2 contains definitions. Section 3 describes the scdawg for linear strings while Section 4 describes its extension to circular strings. Section 5 deals with the computation of occurrences of displayable entities. Section 6 introduces the notion of conflicts and Section 7 lists other queries that are to be implemented. Section 6 also explains how the algorithms implementing queries for linear strings can be modified so that they work with circular strings. Finally, Section 8 mentions some applications for the visualization and analysis of circular strings.

2 Definitions

Let s denote a circular string of size n consisting of characters from a fixed alphabet, Σ, of constant size. Figure 1 shows an example circular string of size 8. We shall represent a circular string by a linear string enclosed in angle brackets "<>" (this distinguishes it from a linear string) . The linear string is obtained by traversing the circular string in clockwise order and listing each element as it is traversed. The starting point of the traversal is chosen arbitrarily. Consequently, there are up to n equivalent representations of s. In the example, s could be represented as $<abcdabce>$, $<bcdabcea>$, etc.

We characterize the relationship between circular strings and linear strings by defining the functions, *linearize* and *circularize*. *linearize* maps circular strings to linear strings. It is a one-many mapping as a circular string can, in general, be mapped to more than one

linear string. For example, *linearize*($<abcd>$) = $\{abcd, bcda, cdab, dabc\}$. We will assume, for the purpose of this paper, that *linearize* arbitrarily chooses one of the linear strings; for convenience we assume that it chooses the representation obtained by removing the angle brackets "<>". So, *linearize*($<abcd>$) = $abcd$. *circularize* maps linear strings to circular strings. It is a many-one function and represents the inverse of *linearize*.

We use lower case letters to represent circular strings and upper case letters to represent linear strings. Further, if a lower case letter (say, s) is used to represent a particular circular string, then the corresponding upper case letter (S) is assumed to be *linearize*(s). A single character in s or S occurring in the ith position is denoted by s_i or S_i, respectively. A substring of S is denoted by $S_{i,j}$ where $i \leq j$. $S_{i,j} = S_i S_{i+1} ... S_j$. A substring of s is denoted by $s_{i,j}$, where $s_{i,j} = S_{i,j}$ if $i \leq j$ and $S_{i,n} S_{1,j}$ if $i > j$. For example, if $s = < abcdabce >$, then $S = abcdabce$. $S_5 = s_5 = a$. $S_{3,5} = s_{3,5} = cda$. $s_{7,2} = ceab$. We use the symbol, γ, to denote either a circular string or a linear string. In the example, $\gamma_{3,5} = S_{3,5}$, if $\gamma = S$; $\gamma_{3,5} = s_{3,5}$, if $\gamma = s$.

The *predecessor, pred(γ,i,j) of a substring $\gamma_{i,j}$ of γ* is defined as

$$pred(\gamma,i,j) = \begin{cases} \gamma_{i-1} & \text{if } 1 < i \leq n \\ \infty & \text{if } i = 1 \text{ and } \gamma \text{ is linear} \\ \gamma_n & \text{if } i = 1 \text{ and } \gamma \text{ is circular} \end{cases}$$

The *successor, succ(γ,i,j) of a substring $\gamma_{i,j}$ of γ* is defined as

$$succ(\gamma,i,j) = \begin{cases} \gamma_{j+1} & \text{if } 1 \leq j < n \\ \infty & \text{if } j = n \text{ and } \gamma \text{ is linear} \\ \gamma_1 & \text{if } j = n \text{ and } \gamma \text{ is circular} \end{cases}$$

The *immediate context, context(γ,i,j) of a substring $\gamma_{i,j}$ of γ* is the ordered pair $(pred(\gamma, i, j), succ(\gamma, i, j))$.

The *predecessor*, $pred(\gamma, \alpha)$, and successor, $succ(\gamma, \alpha)$, sets of a pattern, α, in a string γ are defined as below:
$pred(\gamma, \alpha) = \{pred(\gamma, i, j) | \ \gamma_{i,j} = \alpha \ \}$. $succ(\gamma, \alpha) = \{succ(\gamma, i, j) | \ \gamma_{i,j} = \alpha \ \}$.

The *immediate context set, context(γ,α) of a pattern, α, in γ* is the set $\{context(\gamma, i, j) |\ \gamma_{i,j} = \alpha\ \}$.

In the example string of Figure 1, $succ(s, abc) = succ(S, abc) = \{d, e\}$. $pred(s, abc) = \{d, e\}$; $pred(S, abc) = \{\infty, d\}$. $context(s, abc) = \{(e, d), (d, e)\}$. $context(S, abc) = \{(\infty, d), (d, e)\}$.

A pattern occurring in γ is said to be *maximal* iff its occurrences are not all preceded by the same character nor all followed by the same character. So, a pattern α of length $< n$ in γ is maximal iff $|pred(\gamma, \alpha)| \geq 2$ and $|succ(\gamma, \alpha)| \geq 2$. This is not necessarily true for patterns of length greater than or equal to n. For example, S is maximal in S (since it is neither preceded nor followed by a character), but $|pred(S, S)| = |succ(S, S)| = 1$.

A pattern is said to be a *displayable entity* (or displayable) of γ iff it is maximal and occurs at least twice in γ. Note that if γ represents a circular string, then a pattern can be arbitrarily long. In the rest of our discussion, we will assume that displayable entities of circular strings have length less than n.

3 Scdawgs For Linear Strings

An scdawg, $SCD(S) = (V(S), R(S), L(S))$ corresponding to a string S is a directed acyclic graph defined by a set of vertices, $V(S)$, a set, $R(S)$, of labeled directed edges called right extension (re) edges, and a set of labeled directed edges, $L(S)$ called left extension (le) edges. Each vertex of $V(S)$ represents a substring of S. Specifically, $V(S)$ consists of a source (which represents the empty word, λ), a sink (which represents S), and a vertex corresponding to each displayable entity of S.

Let $de(v)$ denote the string represented by vertex, v, $v \in V(S)$. Define the *implication*, $imp(S, \alpha)$, of a string, α of S to be the smallest superword of α in $\{de(v) |\ v \in V(S)\}$, if such a superword exists. Otherwise, $imp(S, \alpha)$ does not exist. Re edges from v_1 ($v_1 \in V(S)$) are obtained as follows: for each letter, x, in Σ, if $imp(S, de(v_1)x)$ exists and is equal to $de(v_2) = \beta de(v_1)x\gamma$, then there is an re edge from v_1 to v_2 with label $x\gamma$. If β is the empty string, then the edge is known as a *prefix extension edge*. Le edges from v_1 ($v_1 \in V(S)$) are obtained as follows: for each letter, x, in Σ, if $imp(S, xde(v_1))$ exists and is equal to $de(v_2) =$

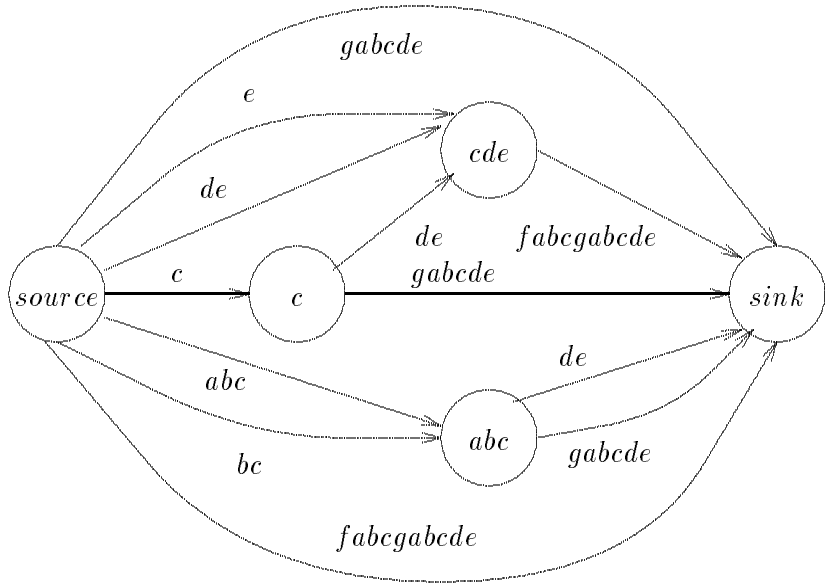


Figure 2: SCDAWG for $S = cdefabcgabcde$, only re edges are shown

$\gamma xde(v_1)\beta$, then there is an le edge from v_1 to v_2 with label γx. If β is the empty string, then the edge is known as a *suffix extension edge*. Figure 2 shows $(V(S),R(S))$ corresponding to $S = cdefabcgabcde$. abc, cde, and c are the displayable entities of S. There are two re edges from the vertex representing abc. These correspond to $x = d$ and $x = g$. $imp(S,abcd)$ $= imp(S,abcg) = S$. Consequently, both edges are incident on the sink. There are no edges corresponding to the other letters of the alphabet as $imp(S,abcx)$ does not exist for $x \in \{a, b, c, e, f\}$.

Notice that the number of re edges from a vertex, v, equals $|succ(S, de(v))$ - $\{\infty\}|$ and the number of le edges equals $|pred(S, de(v))$ - $\{\infty\}|$. In the example, $succ$(S,cde) = $\{\infty, f\}$. So, the number of right edges leaving the vertex corresponding to it is 1.

The space required for $SCD(S)$ is $O(n)$ and the time needed to construct it is $O(n)$ [7, 6]. While we have defined the scdawg data structure for a single string, it can be extended to represent a *set* of strings [6].

4 Extension to Circular Strings

In Section 4.1, we present a constructive definition of an scdawg for circular strings. Section 4.2 analyzes the complexity of the algorithm of Section 4.1 to construct the scdawg of a circular string and Section 4.3 identifies and proves some properties of this scdawg.

4.1 SCDAWGs For Circular Strings

The notion of an scdawg may be extended to circular strings. The scdawg for circular strings is defined constructively by the algorithm of Figure 3. The scdawg for the circular string s is obtained by first constructing the scdawg for the linear string $T = SS$ (recall that $S = linearize(s)$). A bit is associated with each re edge in $R(T)$ indicating whether it is a prefix extension edge or not. Similarly, a bit is associated with each le edge in $L(T)$ to identify suffix extension edges. Two pointers, a suffix pointer and a prefix pointer are associated with each vertex, v in $V(T)$. The suffix (prefix) pointer points to a vertex, w, in $V(T)$ such that $de(w)$ is the largest suffix (prefix) of $de(v)$ represented by any vertex in $V(T)$. Suffix (prefix) pointers are the reverse of suffix (prefix) extension edges and are derived from them. Figure 4 shows $SCD(T) = SCD(SS)$ for $S = cabcbab$. The broken edge from vertex c to vertex abc is a suffix extension edge, while the solid edge from vertex ab to vertex abc is a prefix extension edge.

Next, in step 2, suffix and prefix redundant vertices of $SCD(T)$ are identified. A suffix (prefix) redundant vertex is a vertex v that satisfies the following properties:
(a) v has exactly one outgoing re (le) edge.
(b) $|de(v)| < n$.
A vertex is said to be *redundant* if it is either prefix redundant or suffix redundant or both. In Figure 4, vertex c is prefix redundant only, while vertex ab is suffix redundant only. No other vertices in the figure are redundant (in particular, the vertex representing S is not redundant even though it has one re and one le out edge as $|S| = n$). The fact that step 2 does, in fact, identify all redundant vertices is established later.

Vertices of $SCD(T)$ are processed in reverse topological order in step 3 and redundant

Algorithm A
Step1: Construct $SCD(T)$ for $T = SS$.

Step2(a):
{Identify Suffix Redundant Vertices}
v:= *sink*;
while $v \neq$ *source* **do**
  begin
    v:= v.*suffix*;
    if v has exactly one outgoing re edge
    then
      if $(|de(v)| < n)$
      then mark v suffix redundant;
    else
      exit Step 2(a);
  end;

Step2(b):
{Identify Prefix Redundant vertices}
{Similar to Step 2 (a)}

Step3:
v:= *sink*;
while (v <> *source*) **do**
  begin
    case v **of**
      suffix redundant but not prefix redundant: *ProcessSuffixRedundant(v)*;
      prefix redundant but not suffix redundant: *ProcessPrefixRedundant(v)*;
      suffix redundant and prefix redundant : *ProcessBothRedundant(v)*;
      not redundant : {Do nothing };
    endcase;
    v:= *NextVertexInReverseTopologicalOrder*;
  end;

Figure 3: Algorithm for constructing the scdawg for a circular string

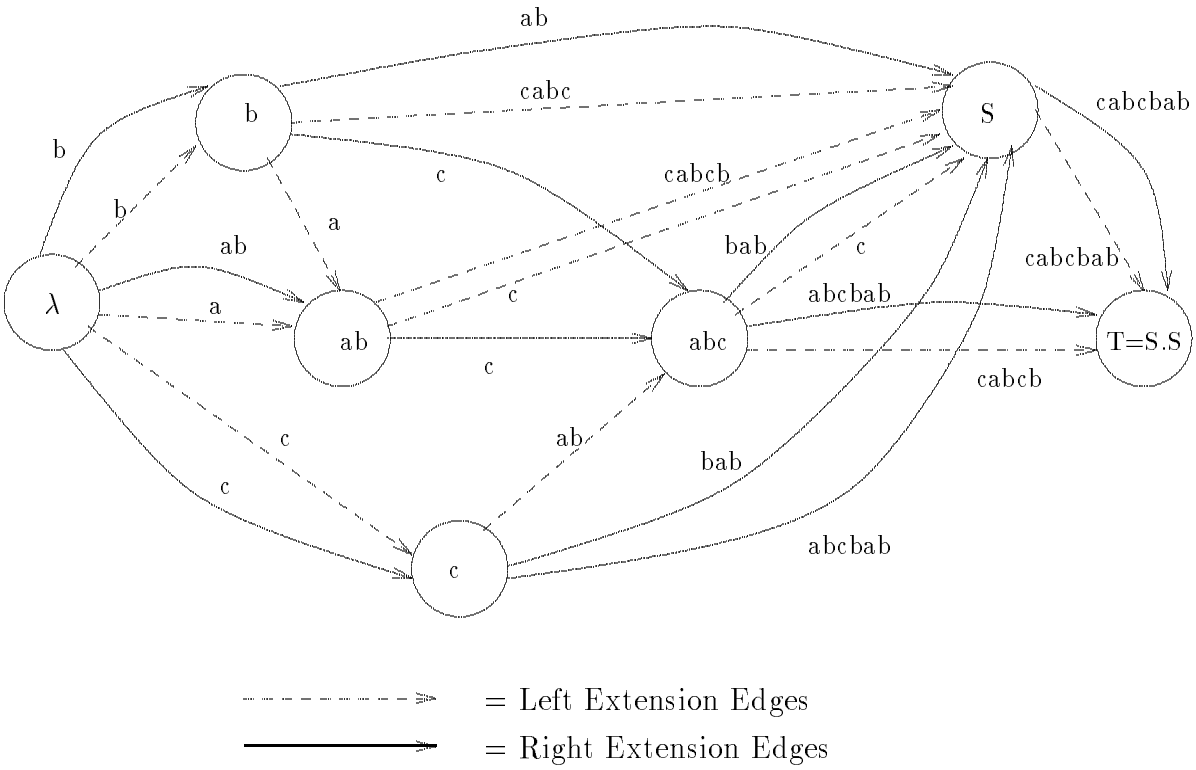


Figure 4: $SCD(T)$ for T=*cabcbabcabcbab*

Procedure ProcessSuffixRedundant(v)

1. Eliminate all left extension edges leaving v (there are at least two of these).
2. There is exactly one right extension edge, e, leaving v. Let the vertex that it leads to be w. Let the label on the right extension edge be $x\gamma$. Delete the edge.
3. All right edges incident on v are updated so that they point to w. Their labels are modified so that they represent the concatenation of their original labels with $x\gamma$.
4. All left edges incident on v are updated so that they point to w. Their labels are not modified. However, if any of these were suffix extension edges, the bit which indicates this should be reset as these edges are no longer suffix extension edges.
5. Delete v.

Figure 5: Algorithm for processing a vertex which is suffix redundant

vertices are eliminated. When a vertex is eliminated, the edges incident to/from it are redirected and relabeled as described in Figures 5 to 10. The resulting graph is $CSCD(s)$. The set of vertices of $CSCD(s)$ is denoted by $CV(s)$. The set of right (left) edges of $CSCD(s)$ is denoted by $CR(s)$ ($CL(s)$). Figure 11 shows $CSCD(s)$ for $s = < cabcbab >$. Notice that vertices c and ab have been eliminated and that the two incoming edges to c and the three incoming edges to ab of Figure 4 now point to abc.

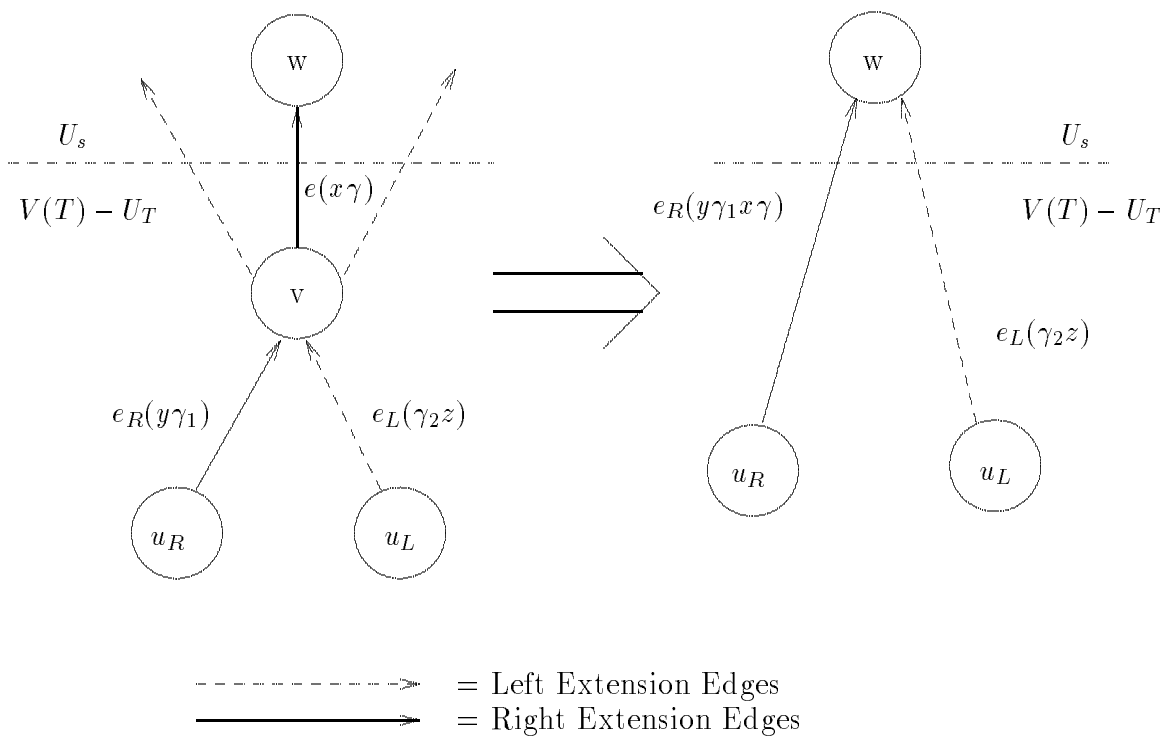


Figure 6: v is suffix redundant

Procedure ProcessPrefixRedundant(v)

1. Eliminate all right extension edges leaving v (there are at least two of these).
2. There is exactly one left extension edge, e, leaving v. Let the vertex that it leads to be w. Let the label on the left extension edge be γx. Delete the edge.
3. All left edges incident on v are updated so that they point to w. Their labels are modified so that they represent the concatenation of γx with their original labels.
4. All right edges incident on v are updated so that they point to w. Their labels are not modified. However, if any of these were prefix extension edges, the bit which indicates this should be reset as these edges are no longer prefix extension edges.
5. Delete v.

Figure 7: Algorithm for processing a vertex which is prefix redundant

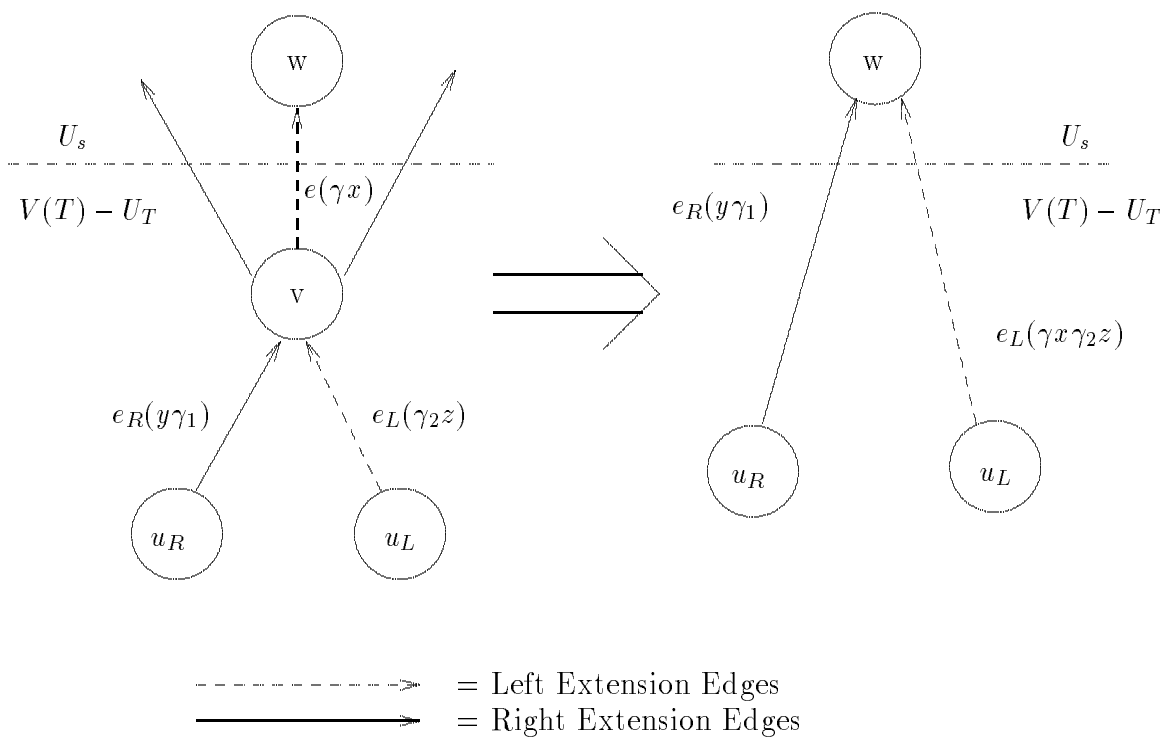


Figure 8: v is prefix redundant

Procedure ProcessBothRedundant(v)

1. There is exactly one right extension edge, e_1, leaving v. Let the vertex that it leads to be w_1. Let the label on the edge be $x\gamma$. Delete the edge.

2. There is exactly one left extension edge, e_2, leaving v. Let the vertex that it leads to be w_2. Let the label on the edge be γx. Delete the edge.
   {We establish later that w_1 and w_2 are, in fact, the same vertex.}

3. All right edges incident on v are updated so that they point to w_1. Their labels are modified so that they represent the concatenation with $x\gamma$. If any of these edges were prefix edges, the bit which indicates this should be reset.

4. Similarly, left edges incident on v are updated so that they point to w_2. Their labels are modified so that they represent the concatenation with γx. If any of these edges were suffix extension edges, the bit which indicates this should be reset.

5. Delete v.

Figure 9: Algorithm for processing a vertex which is prefix and suffix redundant

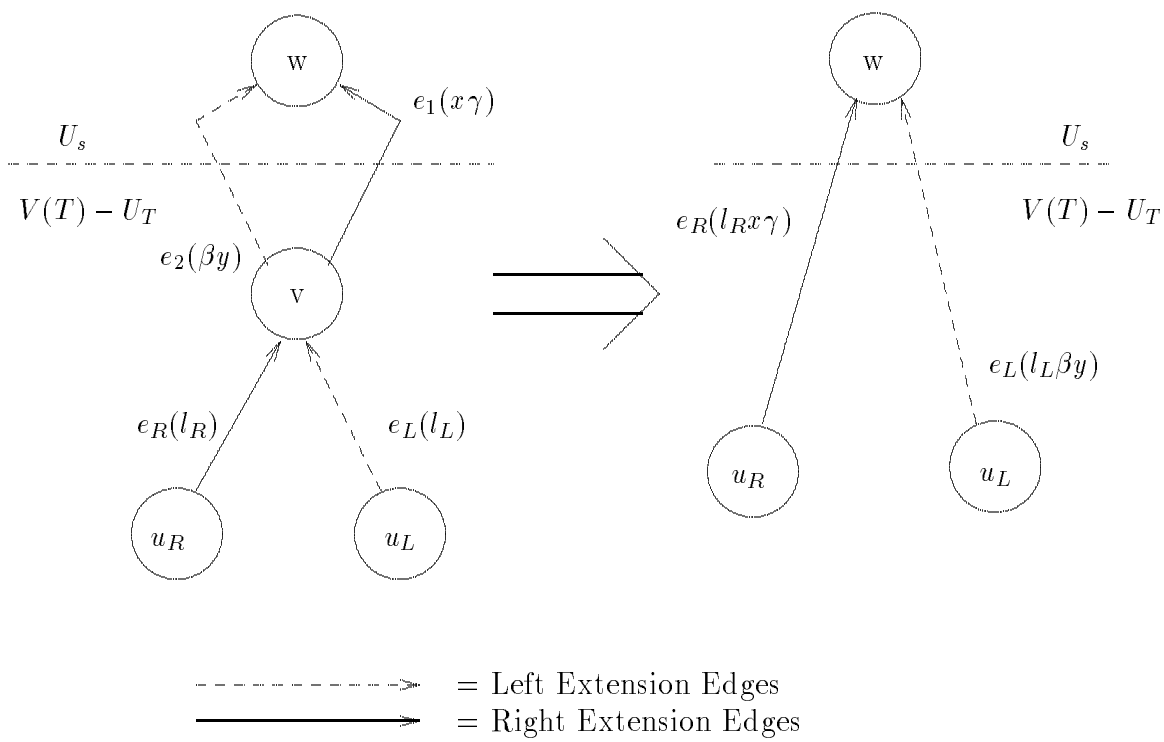


Figure 10: v is suffix and prefix redundant

Lemma 1 *For every substring $s_{i,j}$ of length $< n$ of s, there exists a substring, $T_{l,m}$ $(= s_{i,j})$, of T such that $context(T, l, m) = context(s, i, j)$.*

Proof

Case (i): $i > j$. Clearly, $i \neq 1$, $j \neq n$. By construction, $T_{i,j+n} = s_{i,j}$ and $context(T, i, j+n)$ $= (T_{i-1}, T_{n+j+1}) = (S_{i-1}, S_{j+1}) = context(s, i, j)$.

Case (ii): $i \leq j$. Now, $T_{i,j} = T_{i+n,j+n} = s_{i,j}$.

Subcase (a): $i = 1$, $j \neq n$. $context(T, n+1, n+j) = (T_n, T_{n+j+1}) = (S_n, S_{j+1}) = context(s, i, j)$

Subcase (b): $i \neq 1$, $j = n$. $context(T, i, n) = (T_{i-1}, T_{n+1}) = (S_{i-1}, S_1) = context(s, i, n)$.

Subcase (c): $i \neq 1$, $j \neq n$. $context(T, i, j) = (T_{i-1}, T_{j+1}) = (S_{i-1}, S_{j+1}) = context(s, i, j)$.

Subcase(d): $i = 1$, $j = n$. Not possible since the length of $s_{i,j} < n$. $\square$

Corollary 1 *For every pattern, α, of length $< n$ in s, $context(s, \alpha) \subseteq context(T, \alpha)$.*

Corollary 2 *For every pattern, α, of length $< n$ in s, $pred(s, \alpha) \subseteq pred(T, \alpha)$ and $succ(s, \alpha)$ $\subseteq succ(T, \alpha)$.*

Lemma 2 *Let $T_{i,j}$ be a substring of T. If $i \neq 1$, then there is a substring $s_{l,m}$ $(=T_{i,j})$ of s such that $pred(s, l, m) = pred(T, i, j)$. If $i = 1$, $pred(T, i, j) = \infty$.*

Proof If $i = 1$, the result follows from the definition of $pred(T, i, j)$. If $i \neq 1$, choose $s_{l,m}$ so that $l = i$ if $i \leq n$, and $l = i - n$ if $i > n$; $m = j$ if $j \leq n$, and $m = j - n$ if $j > n$; (if the length of $T_{i,j}$ is greater than n, $s_{l,m}$ is assumed to wrap around once). So, $pred(s, l, m)$ $= S_{l-1} = T_{i-1} = pred(T, i, j)$. $\square$

Corollary 3 *For every pattern, α, of length $< n$ in T, $pred(T, \alpha) - \{\infty\} \subseteq pred(s, \alpha)$.*

Theorem 1 *For every pattern α of length less than n, $pred(s, \alpha) = pred(T, \alpha)$ - $\{\infty\}$ and $succ(s, \alpha) = succ(T, \alpha)$ - $\{\infty\}$.*

Proof From Corollary 2 we have $pred(s, \alpha) \subseteq pred(T, \alpha)$. So, $pred(s, \alpha)$ - $\{\infty\} \subseteq$ $pred(T, \alpha)$ - $\{\infty\}$ and hence $pred(s, \alpha) \subseteq pred(T, \alpha)$ - $\{\infty\}$ (since $pred(s, \alpha)$ does not contain

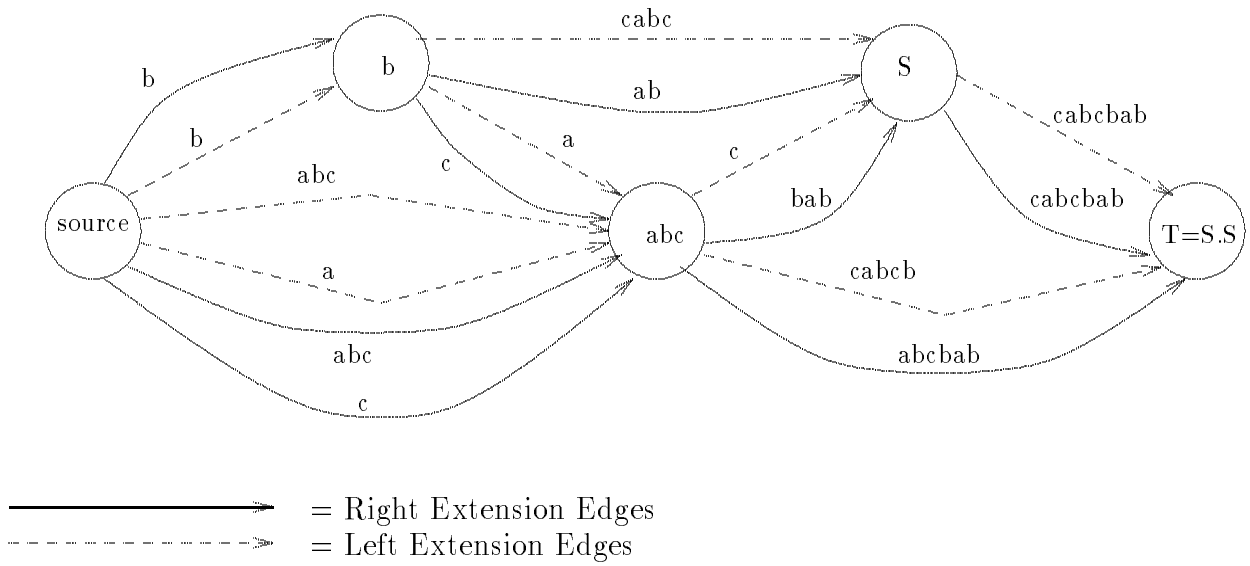


Figure 11: Scdawg for < *cabcbab* >

∞). From Corollary 3 we have $pred(s,\alpha) \supseteq pred(T,\alpha)$ - $\{\infty\}$. So, $pred(s,\alpha) = pred(T,\alpha)$ - $\{\infty\}$. The proof that $succ(s,\alpha) = succ(T,\alpha)$ - $\{\infty\}$ is similar. □

Theorem 2 *A vertex, v with $|de(v)| < n$ in $V(T)$ is non redundant iff $de(v)$ is a displayable entity of s.*

Proof Suppose α is a displayable entity of s. Then, we have $|pred(s,\alpha)| \geq 2$ and $|succ(s,\alpha)| \geq 2$. From Theorem 1 we have $|pred(T,\alpha) - \{\infty\}| \geq 2$ and $|succ(T,\alpha) - \{\infty\}| \geq 2$. So, α is a displayable entity in T and the corresponding vertex in $V(T)$ has at least two le and two re edges leaving it. Hence, v is not redundant.

Next, suppose there is a non redundant vertex, v, in $SCD(T)$ with $|de(v)| < n$. Let $\alpha = de(v)$. Since v is not redundant, $|pred(T,\alpha)| - \{\infty\}| \geq 2$ and $|succ(T,\alpha)| - \{\infty\}| \geq 2$. From Theorem 1 we have $|pred(s,\alpha)| \geq 2$ and $|succ(s,\alpha)| \geq 2$. So, α is a displayable entity of s. □

Corollary 4 *A redundant vertex in $V(T)$ is not a displayable entity of s.*

Lemma 3 *(a) A vertex, v, in $V(T)$ will have exactly one re (le) out edge only if $de(v)$ is a suffix (prefix) of T.*
(b) If a vertex, v, such that $de(v)$ ($|de(v)| < n$) is a suffix (prefix) of T has more than one re (le) out edge, then no vertex, w, such that $de(w)$ is a suffix (prefix) of $de(v)$ can be suffix (prefix) redundant.

Proof (a) Suppose $de(v)$ is not a suffix of T. Then ∞ is not an element of $succ(T, de(v))$. So, $|succ(T, de(v)) - \{\infty\}| = |succ(T, de(v))| \geq 2$. So, v has at least two re out edges, which is a contradiction. Hence, $de(v)$ must be a suffix of T.
(b) Since $de(w)$ is a suffix of $de(v)$, a successor of $de(v)$ must also be a successor of $de(w)$. So, $succ(T, de(w)) - \{\infty\} \supseteq succ(T, de(v)) - \{\infty\}$ or $|succ(T, de(w)) - \{\infty\}| \geq |succ(T, de(v)) - \{\infty\}| \geq 2$ ($de(v)$ has at least two re out edges). So, w must have at least two re out edges and cannot be suffix redundant. $\square$

We can now show that step 2(a) of Algorithm A identifies all suffix redundant vertices in $V(T)$. Since it is sufficient to examine vertices corresponding to suffixes of T (Lemma 3(a)), step 2(a) follows the chain of suffix pointers starting from the sink. If a vertex on this chain representing a displayable entity of length $< n$ has one re out edge, then it is marked suffix redundant. The traversal of the chain terminates either when the source is reached or a vertex with more than one re out edge is encountered (Lemma 3(b)). Similarly, step 2(b) identifies all prefix redundant vertices in $V(T)$.

4.2 Complexity Analysis

Step 1 takes $O(n)$ time [6]. Step 2 will in the worst case traverse all the vertices in $SCD(T)$ spending $O(1)$ time at each. The number of vertices is bounded by $O(n)$ [6]. So, step 2 takes $O(n)$ time. Step 3 traverses $SCD(T)$. Each vertex is processed once; each edge is processed at most twice (once when it is an incoming edge to the vertex being currently processed, and once when it is the out edge from the vertex currently being processed. So, Step 3 takes $O(n)$ time (note that $SCD(T)$ has $O(n)$ edges).

4.3 Properties of $CSCD(s)$

Define the *implication*, $imp(s, \alpha)$, of a string, α, with respect to $CSCD(s)$ to be the smallest superword, $\beta\alpha\gamma$, of α represented by a vertex in $CV(s)$, such that there does not exist a substring $\beta_1\alpha\gamma_1$ of T where the length of the least common suffix, $lcs(\beta, \beta_1)$, of β and β_1 is less than $min(|\beta|, |\beta_1|)$ or the length of the least common prefix, $lcp(\gamma, \gamma_1)$, of γ and γ_1 is less than $min(|\gamma|, |\gamma_1|)$, if such a superword exists. Otherwise, $imp(s, \alpha)$ does not exist. The additional condition (which is referred to as the *uniqueness* condition) that is imposed on $imp(s, \alpha)$ is guaranteed for $imp(T, \alpha)$ by the definition of $SCD(T)$.

Let $R = \{$ *abcaaaa, babcaa, cabcaa* $\}$ be the smallest set of superword displayable entities of *abc* in s such that any superword displayable entity of *abc* in s is a superword of an element of R. Then, $de(s, abc)$ must be one of the elements of R. We have $|lcs(b, c)| = 0 < min(|b|, |c|)$. So, $de(s, abc)$ is neither *babcaa* nor *cabcaa*. Further, since $|lcs(aaaa, aa)| = min(|aaaa|, |aa|)$, $|lcp(b, \lambda)| = min(|b|, |\lambda|)$, and $|lcp(c, \lambda)| = min(|c|, |\lambda|)$, $de(s, abc) =$ *abcaaaa*.

Lemma 4 *Let v be a suffix and prefix redundant vertex in $SCDINT(T)$, where $SCDINT(T)$ represents an intermediate configuration between $SCD(T)$ and $CSCD(s)$ just after the while statement in Step 3 of Algorithm A. Let the le and re out edges be incident on w_1 and w_2 respectively, where $de(w_1) = imp(s, de(v)x) = \beta_1 de(v)x\gamma_1$ and $de(w_2) = imp(s, yde(v) = \beta_2 yde(v)\gamma_2$. If w_1 and w_2 are not redundant, then $w_1 = w_2$.*

Proof **Case 1.** $|de(w_1)| < n$, $|de(w_2)| < n$.
β_1 cannot be *nil* (if it is, then w_1 is prefix redundant since $|de(w_1)| < n$ and all occurrences of $de(v)$ except the prefix of S are preceded by y). Similarly, $\gamma_2 \neq nil$. So, $de(w_1)$ must be of the form $\beta_3 yde(v)x\gamma_1$, since y is the only letter that precedes $de(v)$. Similarly, $de(w_2)$ must be of the form $\beta_2 yde(v)x\gamma_3$. We now show that $\beta_3 = \beta_2$ and $\gamma_1 = \gamma_3$. Assume that this is not the case. Since $|de(w_1)| < n$, $|de(w_2)| < n$ and w_1 and w_2 are not redundant, $|pred(s, de(w_1))|$, $|succ(s, de(w_1))|$, $|pred(s, de(w_2))|$, and $|succ(s, de(w_2))|$ are all at least 2. So, there must exist a displayable entity, $\beta_m yde(v)x\gamma_m$, of s where β_m is the largest common suffix of β_3 and β_2 and γ_m is the largest common prefix of γ_1 and γ_3. Further, $\beta_m yde(v)x\gamma_m$

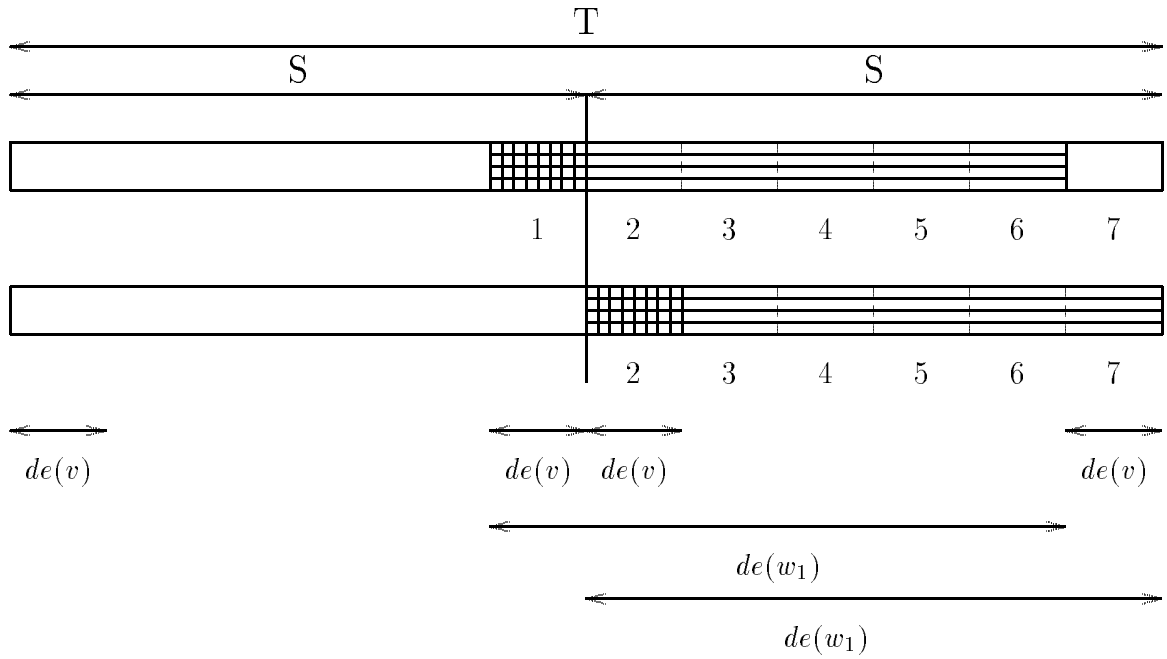


Figure 12: Illustration of proof of prefix/suffix redundancy invariant

$= imp(s, de(v)x) = imp(s, yde(v))$, which contradicts statements made above.

Case 2. $|de(w_1)| \geq n$, $|de(w_2)| < n$.

γ_2 cannot be *nil*, otherwise w_2 is suffix redundant. So, $de(w_2) = \beta_2 yde(v)x\gamma_3$ as x is the only letter that follows $de(v)$. Arguments similar to those in Case 1 show that since $|de(w_2)| < n$ and w_2 is not redundant, γ_1 must be a prefix of γ_3 and β_1 a suffix of $\beta_2 y$. But, then $|de(w_2)| > |de(w_1)| > n$, which is a contradiction. Hence, Case 2 cannot exist.

Case 3. $|de(w_2)| \geq n$, $|de(w_1)| < n$.

Similar to Case 2.

Case 4. $|de(w_2)| \geq n$, $|de(w_1)| \geq n$.

Figure 12 shows that for this case to occur, $S = \alpha^m$, for some α, where $|\alpha| \leq |de(v)|$. Call this the *prefix/suffix redundancy invariant*. The figure assumes that $|de(w_1)| = n$, that $de(v)$ is a prefix of $de(w_1)$, and that $|de(v)| < n/2$ and divides n. However, the prefix/suffix redundancy invariant can be shown to be true in all other cases. Two copies of T are shown in the figure. The first copy shades the occurrence $(n - |de(v)| + 1, n)$ of $de(v)$ and its

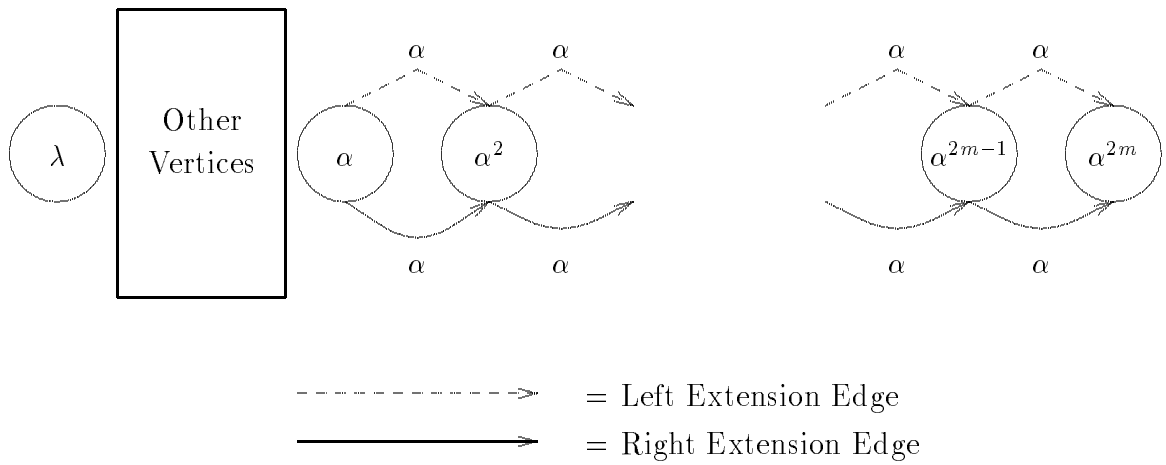


Figure 13: $SCD(\alpha^{2m})$

extension to $de(w_1)$. The second shades the occurrence $(n+1, n+|de(v)|)$ and its extension to $de(w_1)$. Since the shaded regions in both strings represent $de(w_1)$, we have: box 1 = box 2; box 2 = box 3;...;box 6 = box 7. Or, box 1 = box 2 = ... = box 7, and $S = (de(v))^6$.

Next, we assume without loss of generality that there is no β such that $S = \beta^k$, $k > m$. Call this the *smallest repetition assumption.*

The only occurrences of α in T are at $((1, |\alpha|), (|\alpha|+1, 2|\alpha|), ..., ((m-1)\alpha+1, 2n))$ (if not, an argument similar to the one of Figure 12 contradicts the smallest repetition assumption). So, $SCD(T)$ takes the form of Figure 13. Each vertex representing α^i, $1 \leq i \leq 2m-1$, has exactly one le and one re out edge as shown.

All remaining displayable entities of T are subwords of α^2 and are of size less than $|\alpha|$ (if not, an argument identical to the one in Figure 12 contradicts the smallest repetition assumption). The vertices representing these displayable entities are represented by the box in Figure 13.

None of the vertices in the box has out edges incident on vertices representing the displayable entities $\{\alpha^3, \alpha^4, ..., \alpha^{2m}\}$. In particular, no out edges from the vertices in the box are incident on vertices representing displayable entities of length greater than n. After

$SCD(\alpha^{2m})$ has been processed by *Algorithm A*, all incoming edges to vertices corresponding to α and α^2 in $SCD(\alpha^{2m})$ are incident on the vertex corresponding to $S = \alpha^m$ in $CSCD(\alpha^{2m})$. It follows that any prefix and suffix redundant vertex in $SCD(\alpha^{2m})$, when processed by Step 3 of *Algorithm A* can have both edges incident on w_1 and w_2 such that $|de(w_1)|$ and $|de(w_2)|$ are at least n only if $de(w_1) = de(w_2) = n$. $\square$

$CSCD(s)$ satisfies properties P1, P2, and P3 stated below (Theorem 3). These properties ensure that the algorithms of [5] can be extended to circular strings.

$P1$: $CV(s)$ consists of a *source* and a *sink*. For each v of $CV(s)$ that is not the source or sink, the following are true:

(a) $|de(v)| < n$ iff $de(v)$ is a displayable entity of s.

(b) if $|de(v)| \geq n$, then $de(v)$ is a displayable entity of T.

$P2$: There exists an re out edge corresponding to letter x in Σ from vertex v_1 in $CV(s)$ to vertex v_2 in $CV(s)$ iff $imp(s, de(v_1)x)$ exists and is equal to $de(v_2)$. If $de(v_2) = \beta de(v_1)x\gamma$, then the label on the re edge is $x\gamma$. If $\beta = nil$, then the edge is a prefix extension edge.

$P3$: Similar to $P2$ but for le edges.

Theorem 3 *$CSCD(s)$ satisfies $P1$, $P2$, and $P3$.*

Proof Property $P1$ is established by the knowledge that $SCD(T)$ contains all displayable entities of T and that *Algorithm A* only eliminates those displayable entities of T of length less than n, which are not displayable entities of s (Corollary 4).

$P2$ and $P3$ are proved by induction. The induction hypothesis is:

Let U_s be the subset of U_T that remains after the vertex set $U_T \subseteq V(T)$ has been processed by step 3 of *Algorithm A*.

(I) Let R_{U_s} be the set of re edges which are incident on vertices in U_s. For any re edge $r \in R_{U_s}$ from vertex u to w with label $x\gamma$, $imp(s, de(u)x) = de(w) = \beta de(u)x\gamma$. An analogous condition holds for le edges.

(II) For each vertex u in $U_s \cup (V(T) - U_T)$, there is an re out edge corresponding to each letter x in $succ(T, de(u)) - \{\infty\}$ incident on a vertex in $U_s \cup (V(T) - U_T)$. An analogous condition holds for le edges.

When $U_T = V(T)$, we have $U_s = CV(s)$, by definition. So, $R_{CV(s)} = CR(s)$. (I) establishes that these edges are incident on the correct vertices and that their labels are correct. (II) establishes that $CR(s)$ is complete. So $P2$ holds. Similarly, $P3$ holds.

Induction Base: $U_T = U_s = \{\}$. R_{U_s} and L_{U_s} are empty so (I) does not apply. (II) is established from the definition of $SCD(T)$.

Induction Step: Consider vertex, v ($v \epsilon V(T)$), which is about to be processed by step 3 of algorithm A. Let U'_T and U'_s denote U_T and U_s respectively after v has been processed. We must show that (I) and (II) hold for U'_s and U'_T. Since the vertices are processed in reverse topological order, all out edges from v are incident on vertices in U_s and are therefore elements of R_{U_s} or L_{U_s}. So, they must satisfy (I).

Case 1: v is not redundant. $U'_T = U_T \cup \{v\}$; $U'_s = U_s \cup \{v\}$ since v is not eliminated. We must show that (I) is true for incoming edges to v as these are the only additions to R_{U_s} and L_{U_s}. I.e., $R_{U'_s} = R_{U_s} + \{$incoming right edges to $v\}$, $L_{U'_s} = L_{U_s} + \{$incoming left edges to $v\}$.

Let e be an re edge with label $x\gamma$ from u to v. From the definition of $SCD(T)$, we have $de(v) = imp(T, de(u)x) = \beta de(u)x\gamma$, for some β. $imp(T, de(u)x)$ is the smallest superword of $de(u)x$ in $\{de(w)|w \epsilon V(T)\}$. Since $CV(s) \subseteq V(T)$, $\{de(w)|w \epsilon CV(s)\} \subseteq \{de(w)|w \epsilon V(T)\}$ and $imp(s, de(u)x) = imp(T, de(u)x)$ iff $imp(T, de(u)x) \epsilon \{de(w)|w \epsilon CV(s)\}$. But, this is true since $v \epsilon CV(s)$. So, $de(v) = imp(s, de(u)x) = \beta de(u)x\gamma$ and (I) is satisfied. A symmetric argument can be made for incoming le edges to v.

The letter of the alphabet to which an re (le) out edge corresponds is the first (last) character in its label. Since no out edges are added, deleted, or redirected and the labels of all out edges are unchanged, each vertex has an re/le out edge corresponding to the same letter of the alphabet as it had prior to processing vertex v. So, (II) holds (induction hypothesis).

Case 2: v is redundant. $U'_T = U_T \cup \{v\}$. $U'_s = U_s$, since v is eliminated.

Subcase (a): v is suffix redundant only. By definition, v consists of a single re out edge, e, to a vertex w in U_s. Let $label(e) = x\gamma$. From the induction hypothesis,

$imp(s, de(v)x) = de(w) = \beta de(v)x\gamma$. We first establish that (i) $de(w) = imp(s, de(v))$ and (ii) $de(v)$ is a prefix of $de(w)$.

$imp(s, de(v)) \neq de(v)$ as v is redundant. So, $imp(s, de(v))$ must correspond to a vertex on which one of the out edges from v is incident, since there is an out edge corresponding to each element in $pred(s, de(v)) \cup succ(s, de(v))$ (from (II)). The single re edge is incident on w, which represents $imp(s, de(v)x)$. The left out edges from v are incident on vertices which represent $imp(s, x_i de(v))$ for $1 \leq i \leq |pred(s, de(v))| \geq 2$. From the definition of $imp(s, de(v))$, none of these vertices can possibly represent $imp(s, de(v))$. For instance, if $imp(s, x_i de(v))$ is $imp(s, de(v))$, then the string, $imp(s, x_j de(v))$, $i \neq j$, would invalidate the definition.

So, $imp(s, de(v))$ must be $de(w)$. However, for this to be true, we must show that $\beta = nil$ and therefore that $de(v)$ is a prefix of $de(w)$. All occurrences of $de(v)$ in s are followed by x. So, $|pred(T, de(v)x) - \{\infty\}| = |pred(s, de(v)x)| = |pred(s, de(v))| \geq 2$. An argument similar to the one in the previous paragraph shows that for $imp(s, de(v)x)$ to exist, $\beta = nil$.

We have $R_{U'_s} = R_{U_s}$ - {single re out edge from v} + {incoming re edges to v} and $L_{U'_s} = L_{U_s}$ - {le out edges from v} + {incoming le edges to v}. (I) and (II) do not apply to the edges deleted from R_{U_s} and L_{U_s}. So, we only need to prove (I) and (II) for incoming edges to v.

Let e_R be an re edge incident on v from vertex u_R with label $y\gamma_1$ so that $de(v) = imp(T, de(u_R)y) = \beta_1 de(u_R)y\gamma_1$. e_R must be redirected to $imp(s, de(u_R)y)$ for (I) to hold. $imp(T, de(u_R)y) = de(v)$ is the smallest superword of $de(u_R)y$ in $\{de(a)|\ a \in V(T)\}$. $imp(s, de(u_R)y)$ is the smallest superword of $de(u_R)y$ that satisfies the uniqueness condition in $\{de(a)|$ a $\in CV(s)\} \subseteq \{de(a)|\ a \in V(T)\}$. Since $v \notin CV(s)$, $imp(s, de(u_R)y)$ is the smallest superword of $de(v)$ that satisfies the uniqueness condition in $\{de(a)|\ a \in CV(s)\}$. So $imp(s, de(u_R)y) = imp(s, de(v)) = de(w) = \beta_1 de(u_R)y\gamma_1 x\gamma$. The updated re edge, e_R, is incident on w and has label $y\gamma_1 x\gamma$ which was obtained in step 3 of *Algorithm A* by concatenating $label(e_R)$ with $label(e)$. If $\beta_1 = nil$, then e_R continues to be a prefix extension edge. e_R satisfies (I).

Let e_L be an le edge incident on v from u_L so that $de(v) = imp(T, zde(u_l)) = \gamma_2 zde(u_L)\beta_2$

($label(e_L) = \gamma_2 z$). Using the same argument that was used for e_R, we have $imp(s, zde(u_L))$ $= de(w) = \gamma_2 zde(u_L)\beta_2 x\gamma$. e_L is redirected to w and its label remains unchanged. Clearly, e_L is no longer a suffix edge even if $\beta_2 = nil$, because $x\gamma \neq nil$. So, e_L satisfies (I).

Notice that (II) continues to be satisfied as each out edge corresponding to any vertex in $U'_s \cup (V(T) - U_T)$ continues to be associated with the same character (in particular, $label(e_R)$ continues to begin with y and $label(e_L)$ continues to end with z); and each out edge continues to leave the same vertex (in particular, e_R continues to leave u_R, e_L continues to leave u_L).

Subcase (b): v is prefix redundant only. Symmetric to subcase (a).

Subcase (c): v is prefix and suffix redundant. So, v has one re out edge, e_1, to vertex w_1 in $CV(s)$. Let $label(e_1) = x\gamma_1$. Also, v has one le out edge, e_2, to vertex w_2 in $CV(s)$. Let $label(e_2) = \beta_2 y$.

From the induction hypothesis, $de(w_1) = imp(s, de(v)x) = \beta_1 de(v)x\gamma_1$ and $de(w_2) = imp(s, yde(v) = \beta_2 yde(v)\gamma_2$.

The conditions for Lemma 4 are satisfied since w_1 and w_2 are not redundant (otherwise they would have been eliminated). Thus, $de(w_1) = de(w_2) = de(w)$ (say). $imp(s, de(v))$ can either be $imp(s, de(v)x)$ or $imp(s, yde(v))$. But, both these expressions are equal to $de(w)$. So, $imp(s, de(v)) = de(w)$.

The proof that (I) and (II) are satisfied is similar to that for subcase (a). Note, however, that any incoming prefix/suffix extension edges to v will no longer remain prefix/suffix extension edges as $x\gamma$ and βy are not nil. □

5 Computing Occurrences of Displayable Entities

Procedure $LinearOccurrences(S, v)$ of Figure 14, which is based on the outline in [6], reports the end position of each occurrence of $de(v)$, $v \in V(S)$, in the linear string S . However, invoking $LinearOccurrences(T, v)$, $v \in CV(s)$, does not immediately yield all occurrences of $de(v)$ in T. In Section 5.1 we present a modification which obtains all occurrences of

displayable entities of s. In Section 5.2 we show that this modification is correct and that its time complexity is optimal.

5.1 Algorithm

An auxiliary boolean array , *reported[1..n]*, is used in conjunction with $CSCD(s)$. Initially, all elements of this array are set to *false*. Procedure $CircOccurrences(s, v)$ of Figure 15 computes the end positions of each $de(v)$ ($v \in CV(s)$) in s. $LinearOccurrences(T, v)$ of line 1 will not necessarily compute all occurrences of $de(v)$ in T, since it is being executed on $CSCD(s)$ and not on $SCD(T)$. Note, also, that an occurrence of $de(v)$ ending at position i ($i \leq n$) in T has an identical occurrence ending at position $n + i$ in T (since $T = S.S$). Both these occurrences correspond to the same occurrence of $de(v)$ in s. So, if $LinearOccurrences(T, v)$ reports both occurrences, then only the single corresponding occurrence of $de(v)$ in s must eventually be reported.

Lines 4-7 transform the occurrence l, if necessary, so that it represents a value between 1 and n. If this occurrence has not already been listed, then it is added to the list of occurrences and the corresponding element of *reported* is set to *true*. If the occurrence has been listed then it is a duplicate (lines 8-12). After all occurrences have been computed, all elements of *reported* are reset to *false* (lines 14,15) so that *reported* can subsequently be reused to compute the occurrences of some other displayable entity in s.

In the example of Figure 11, $LinearOccurrences(T, v)$, where v represents *abc*, does report the end positions of all occurrences of *abc* in T (i.e., 4, 8, and 11). Lines 2 to 12 transform this into the list of end positions of *abc* in s (i.e., 1 and 4) corresponding to $s_{6,1}$ and $s_{2,4}$ respectively.

Figure 16 shows the $de(v)$'s, $de(w)$'s, and $de(x)$'s for a hypothetical string $T = SS$. Figure 17 shows some fragments of its scdawg. v is suffix redundant in $SCD(T)$ and its single re out edge is incident on w. There is an re edge from x to v and x is not redundant. By construction, the re edge from x to v in $SCD(T)$ becomes an re edge from x to w in $CSCD(s)$. Procedure $LinearOccurrences(T, x)$, x in $CSCD(s)$ will fail to yield the rightmost occurrence of $de(x)$ in T, since that occurrence is neither a subword of $de(w)$

```
Procedure LinearOccurrences(S:string, v:vertex)
{Obtain all occurrences of de(v), v ϵ V(S), in S}
    Occurrences(S,v,0);


Procedure Occurrences(S:linear string, v:vertex, i:integer)
    begin
        if de(v) is a suffix of S
        then output(|S| − i);
        for each re out edge, e, from v in SCD(S) do
            begin
                let w be the vertex on which e is incident;
                Occurrences(S,w,|label(e)| + i);
            end;
    end;
```

Figure 14: Obtaining all occurrences of a displayable entity in a linear string

```
Procedure CircOccurrences(s:circular string, v:vertex)
{v is a vertex in CSCD(s)}
        LinearOccurrences(linearize(s).linearize(s), v);
        for each reported occurrence l of de(v) do
            begin
                if (l > |s|)
                    k:= l - |s|
                else
                    k := l;
                if not reported[k] then
                    begin
                        add k to final list of occurrences
                        reported[k] := true
                    end
            end
        for each occurrence, l, of de(v) in s do
            reported[l]:= false;
```

Figure 15: Obtaining all occurrences of a displayable entity in a circular string

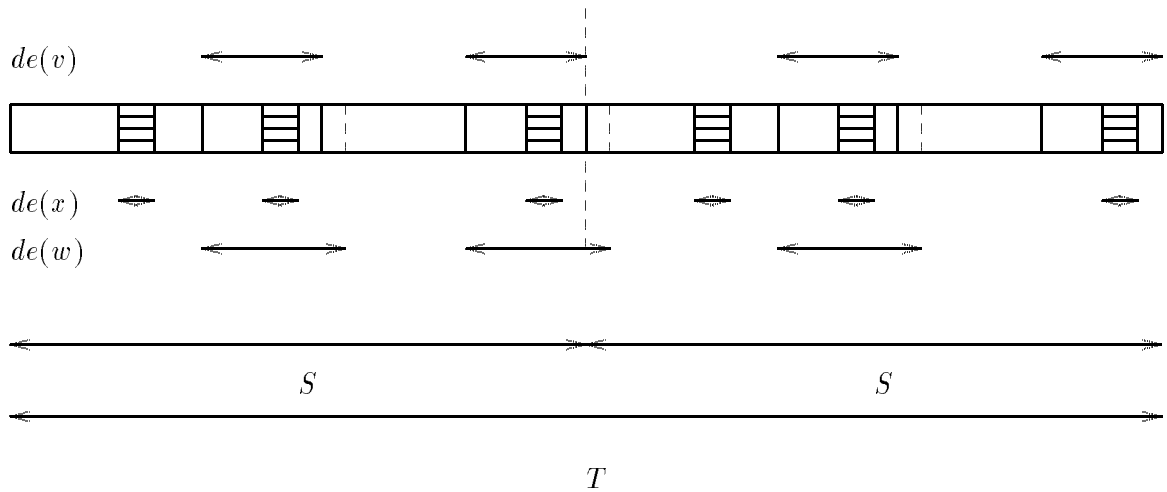


Figure 16: Example string

nor a suffix of T. In the next section, we show that *CircOccurrences*(s, x) computes all occurrences of $de(x)$ in s in spite of the fact that *LinearOccurrences*(T, v) does not compute all occurrences of $de(x)$ in T.

5.2 Proof of Correctness

Let $T_{i,j} = de(v)$, $v \epsilon V(T)$, be a substring of T. Assume that $T_{i,j}$ is not a suffix of T (i.e., $j \neq 2n$). Let $y = imp(T, de(v)T_{j+1}) = \beta de(v)T_{j+1}\gamma$. Then, define the *immediate right extension* $IRE(SCD(T), T_{i,j})$ of $T_{i,j}$ in $SCD(T)$ to be the occurrence $T_{i-|\beta|,j+|\gamma|+1}$ of displayable entity, y.

Let $T_{i,j} = de(v)$, $v \epsilon CV(s)$, be a substring of T. Assume that $T_{i,j}$ is not a suffix of T (i.e., $j \neq 2n$). Let $y = imp(s, de(v)T_{j+1}) = \beta de(v)T_{j+1}\gamma$. Then, define the *immediate right extension* $IRE(CSCD(s), T_{i,j})$ of $T_{i,j}$ in $CSCD(s)$ to be the occurrence $T_{i-|\beta|,j+|\gamma|+1}$ of displayable entity, y.

So, if in Figure 16, $de(v) = \gamma de(x)\alpha$, and $de(w) = de(v)\beta$, then $IRE(SCD(T), T_{2n-|de(x)\alpha|+1,2n-|\alpha|})$ $= T_{2n-|de(v)|+1,2n}$ which is the occurrence of $de(v)$ corresponding to the suffix of T. However, $IRE(CSCD(s), T_{2n-|de(x)\alpha|+1,2n-|\alpha|}) = T_{2n-|de(v)|+1,2n+|\beta|}$ which does not represent a valid substring of T.

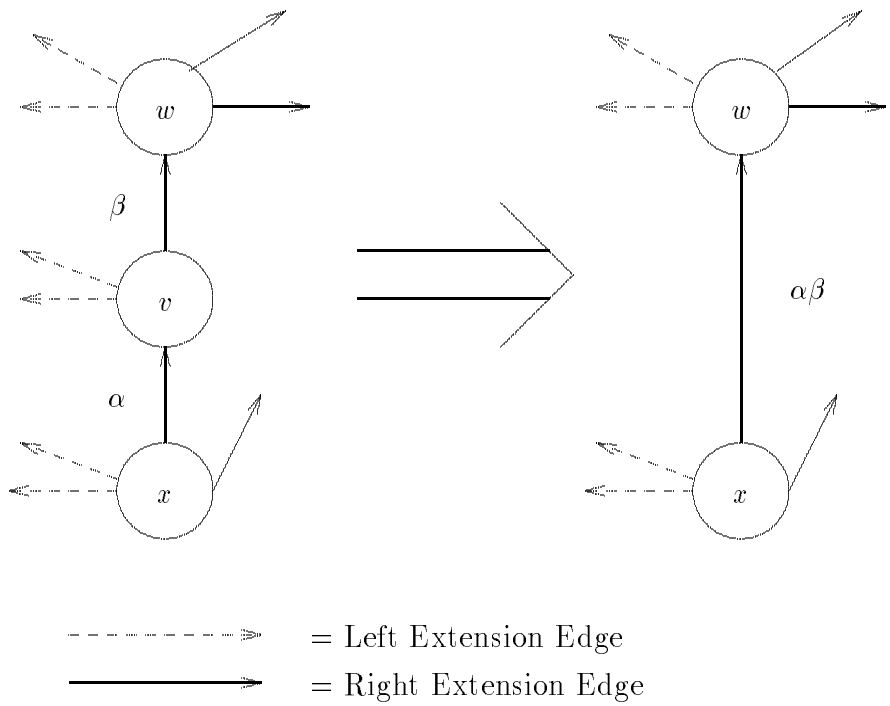


Figure 17: Fragments of scdawgs corresponding to Figure 16

Let $DAWG$ represent either $SCD(T)$ or $CSCD(s)$. Then $IRE^k(DAWG, T_{i,j})$ denotes $IRE(DAWG, IRE^{k-1}(DAWG, T_{i,j}))$ if $k \geq 1$, and $T_{i,j}$ if $k = 0$.

An occurrence $T_{i,j} = de(v)$, $v \epsilon V(T)$ is said to be *Right Retrievable (RR)* in $SCD(T)$ iff one of the following is true:

(i) $j = 2n$.

(ii) $j \neq 2n$ and $IRE(SCD(T), T_{i,j})$ is RR in $SCD(T)$.

Similarly, an occurrence $T_{i,j} = de(v)$, $v \epsilon CV(s)$ is said to be *Right Retrievable (RR)* in $CSCD(s)$ iff one of the following is true:

(i) $j = 2n$.

(ii) $j \neq 2n$ and $IRE(CSCD(s), T_{i,j})$ is RR in $CSCD(s)$.

$IRE(CSCD(s), T_{i,j})$ is defined for any occurrence, $T_{i,j} = de(v)$, $v \epsilon CV(s)$, where $j \neq 2n$. So, $T_{i,j}$ is not RR in $CSCD(s)$ only if (i) $IRE(CSCD(s), T_{i,j})$ does not represent a substring of T or (ii)$IRE(CSCD(s), T_{i,j})$ is a valid substring of T, but is not RR in

$CSCD(s)$.

In the example of Figure 16, $T_{2n-|de(x)\alpha|+1,2n-|\alpha|}$ is RR in $SCD(T)$, but not RR in $CSCD(s)$.

Notice that $(i_p, j_p) = IRE(s,(i,j))$ is not a substring of T iff $i_p < 1$ or $j_p > 2n$.

Lemma 5 *For $k \geq 1$, if $IRE^{k-1}(CSCD(s), T_{i,j})$ and $IRE^{k-1}(CSCD(s), T_{i+n,j+n})$ represent substrings of T and if $(i_p, j_p) = IRE^k(CSCD(s), T_{i,j})$ and $(i_q, j_q) = IRE^k(CSCD(s), T_{i+n,j+n})$, then $i_p + n = i_q$ and $j_p + n = j_q$.*

Proof Assume that there exists a pair of substrings T_{i_1,j_1} and T_{i_2,j_2} of T, such that $i_2 = i_1 + n$ and $j_2 = j_1 + n$ and that $j_1 < n$ (i.e.,we are assuming that their IRE's are defined). By symmetry, both occurrences represent the same displayable entity (say, $de(v)$). Further, $T_{j_1+1} = T_{j_2+1}$ (also by symmetry). Clearly, $imp(s, de(v).T_{j_1+1}) = imp(s, de(v).T_{j_2+1})$. If $(i_3, j_3) = IRE(CSCD(s), T_{i_1,j_1})$ and $(i_4, j_4) = IRE(CSCD(s), T_{i_2,j_2})$, then from the definition of IRE, we have $i_4 = i_3 + n$ and $j_4 = j_3 + n$. Applying this argument repeatedly proves the lemma □

Lemma 6 *The RR occurrences of $de(v)$, v in $V(T)$ ($CV(s)$) in $SCD(T)$ ($CSCD(s)$) are exactly those occurrences of $de(v)$ which are obtained by LinearOccurrences(T,v).*

Proof Follows from the definition of RR occurrences. □

Corollary 5 *All occurrences of a pattern $de(v)$ ($v \in V(T)$) in T are obtained by LinearOccurrences(T, v).*

Lemma 7 *All occurrences of $de(v)$ in T, $v \in CV(s)$, where $|de(v)| \geq n$ are obtained by LinearOccurrences(T,v).*

Proof This follows from Corollary 5 and the construction of $CSCD(s)$ in which no right out edges from vertices representing displayable entities of size $\geq n$ were modified. □

Lemma 8 *All occurrences, $T_{i,j}$, of $de(v)$, where $|de(v)| < n$, $v \in CV(S)$ with $i \leq n$, $j \geq n$ are RR in $CSCD(s)$.*

Proof Assume that the lemma is false and that there exists an occurrence, $T_{i,j}$, of $de(v)$ with $i \leq n$, $j \geq n$ which is not RR in $CSCD(s)$.

Clearly, $j \neq 2n$, otherwise $T_{i,j}$ would be RR in $CSCD(s)$. Let *last* denote the smallest value of k for which $IRE^k(CSCD(s), T_{i,j})$ is not a substring of T. Such a *last* ≥ 1 must exist since $T_{i,j}$ is not RR. Let (i_{last}, j_{last}) denote $IRE^{last}(CSCD(s), T_{i,j})$. Let z be the vertex in $CV(s)$ to which $T_{i_{last}, j_{last}}$ corresponds.

Case 1. $i_{last} < 1$

Clearly, $n < j_{last} < 2n$. Consider the string $T_{1,j_{last}}$ in T. Its length is greater than n. If there were two occurrences of this string in T, then it would be a displayable entity of length $> n$ (because (i) $T_{i,j_{last}}$ does not have a predecessor and (ii) $de(z)$ is maximal and its occurrences are not all followed by the same letter). A vertex corresponding to this displayable entity would not have been eliminated by *Algorithm A* since its length would be $\geq n$ and $T_{1,j_{last}}$ would be RR in $CSCD(s)$ (Lemma 7). So, there must exist only one occurrence of the string represented by $T_{i,j_{last}}$. But, this string is a proper suffix of $de(z)$ which means that one of its occurrences is preceded by a character. So, there *are* two occurrences of this string. This leads to a contradiction.

Case 2. $j_{last} > 2n$

The proof is similar to the one for Case 1. □

Lemma 9 *At least one of the two occurrences, $T_{i,j}$ and $T_{i+n,j+n}$, of $de(v)$, $|de(v)| < n$, $v \in CV(s)$, with $i, j \leq n$ is RR in $CSCD(s)$.*

Proof Assume that the lemma is false. Let *last* be the smallest value of k for which either $IRE^k(CSCD(s), T_{i,j})$ or $IRE^k(CSCD(s), T_{i+n,j+n})$ is not a substring of T. Let $(i_p, j_p) = IRE^{last}(CSCD(s), T_{i,j})$ and $(i_q, j_q) = IRE^{last}(CSCD(s), T_{i+n,j+n})$.

Case 1. $IRE^{last}(CSCD(s), T_{i,j})$ is not a substring of T; $IRE^{last}(CSCD(s), T_{i+n,j+n})$ is a substring of T.

I.e, $i_p < 1$ and $j_q \leq 2n$. So, $j_p \leq n$ and $i_q \leq n$ (from Lemma 5). (i_q, j_q) is RR in $CSCD(s)$, since (i_q, j_q) satisfies the conditions of Lemma 8.

Case 2. $IRE^{last}(CSCD(s), T_{i,j})$ is a substring of T; $IRE^{last}(CSCD(s), T_{i+n,j+n})$ is not a substring of T.

Symmetric to Case 1.

Case 3. $IRE^{last}(CSCD(s), T_{i,j})$ is not a substring of T; $IRE^{last}(CSCD(s), T_{i+n,j+n})$ is not a substring of T.

I.e., $i_p < 1$ and $j_q \geq 2n$. So, $j_p > n$ and $i_q \leq n$ (Lemma 5). This is shown to cause a contradiction by an argument similar to the one in Lemma 8. □

Theorem 4 *Procedure CircOccurrences(s,v) correctly obtains all occurrences of $de(v)$ in s*

.

Proof Lemma 6 shows that $LinearOccurrences(T, v)$ computes all RR occurrences of $de(v)$ in $CSCD(s)$. Lemmas 8 and 9 show that each occurrence of $de(v)$ in s has at least one corresponding occurrence in T, which is RR in $CSCD(s)$. *CircOccurrences* computes these occurrences in T and transforms them so that they represent occurrences in s, removing duplicates if any. So, the output is a list of all occurrences of $de(v)$ in s. □

Theorem 5 *Procedure CircOccurrences is optimal.*

Proof Procedure $CircOccurrences(s, v)$ takes $O(|occ(T, v)|)$ time, where $|occ(T, v)|$ is the number of occurrences of $de(v)$ in T. Each **for** loop takes $O(|occ(T, v)|)$ time.

But, $|occ(T, v)| \leq 2|occ(s, v)|$, where $|occ(s, v)|$ is the number of occurrences of $de(v)$ in s. So, the complexity is $O(|occ(s, v)|)$. $|occ(s, v)|$ is the size of the output, so the algorithm is optimal. □

6 Computing Conflicts Efficiently

[4] defines the concept of *conflicts* and explains its importance in the analysis and visualization of strings. Formally,

(i) A *subword conflict* between two displayable entities, D_1 and D_2, in S exists iff D_1 is a substring of D_2.

(ii) A *prefix-suffix conflict* between two displayable entities, D_1 and D_2, in S exists iff there exist substrings, S_p, S_m, S_s in S such that $S_pS_mS_s$ occurs in S and $S_pS_m = D_1$ and S_mS_s

$= D_2$. The string, S_m is known as the *intersection* of the conflict; the conflict is said to occur between D_1 and D_2 *with respect to* S_m.

[4] also identified a number of problems relating to the computation of conflicts in a linear string, while [5] presented efficient algorithms for most of these problems (some of which are listed in the next section). These algorithms typically involve sophisticated traversals or operations on the scdawg for linear strings. Our extension of scdawgs to circular strings makes it possible to use the same algorithms to solve the corresponding problems for circular strings with some minor modifications which are outlined below.

There are conceptually two kinds of traversals that the algorithms of [5] perform on an scdawg corresponding to a linear string:
(i) Traversal of displayable entities of the string. In these traversals, a vertex is traversed specifically *because* it represents a displayable entity of the string.
(ii) Incidental traversals. In these traversals, a vertex is not traversed because it is a displayable entity, but because it performs some other function. For example, this includes vertices traversed by $LinearOccurrences(T, v)$.

Traversals of type (i) in $CSCD(s)$ are not required to traverse vertices which represent displayable entities of size greater than or equal to n. This may be achieved simply by disabling edges in $CSCD(s)$ which leave a vertex representing a displayable entity of size less than n and are incident on a vertex representing a displayable entity of size greater than or equal to n. Traversals of type (ii), however, may be required to traverse vertices representing displayable entities of size greater than or equal to n. This is achieved by associating a bit for each edge which is set to 1 if it represents an edge from a vertex whose displayable entity is of size less than n to a vertex whose displayable entity is of size greater than or equal to n. Otherwise, it is set to 0. Type (i) traversals check the bit, while type (ii) traversals ignore it.

Finally, all calls to *LinearOccurrences* are replaced by calls to *CircOccurrences*.

7 Other Queries

In this section, we list queries that a system for the visualization and analysis of circular strings would support. [5] contains algorithms for these same queries for linear strings. In the previous section, we showed how these algorithms could be modified to support these queries.

Size Restricted Queries: Experimental data show that random strings contain a large number of displayable entities whose lengths are small. In most applications, small displayable entities are uninteresting. Hence, it is useful to list only those displayable entities whose lengths are greater than some integer, k. Similarly, it is useful to report exactly those conflicts in which the conflicting displayable entities have length greater than k. This gives rise to the following problems:

(1) List all occurrences of displayable entities whose length is greater than k.

(2) Compute all prefix suffix conflicts involving displayable entities of length greater than k.

(3) Compute all subword conflicts involving displayable entities of length greater than k.

An alternative formulation of the problem which also seeks to achieve the goal outlined above is based on reporting only those conflicts whose *size* is greater than k. The size of a conflict is defined below:

The *overlap* of a conflict is defined as the string common to the conflicting displayable entities. The overlap of a subword conflict is the subword displayable entity. The overlap of a prefix-suffix conflict is its intersection. The *size* of a conflict is the length of the overlap.

This formulation of the problem is particularly relevant when the conflicts are of more interest than the displayable entities. It also ensures that all conflicting displayable entities reported have size greater than k. We have the following problems:

(4) Obtain all prefix-suffix conflicts of size greater than some integer k.

(5) Obtain all subword conflicts of size greater than some integer k.

Pattern Restricted Queries: These queries are useful in applications where the fact that two patterns have a conflict is more important than the number or location of the conflicts. The following problems arise as a result:

(6) List all pairs of displayable entities which have subword conflicts.

(7) List all triplets of displayable entities (D_1,D_2,D_m) such that there is a prefix suffix conflict between D_1 and D_2 with respect to D_m.

(8) Same as 6, but size restricted as in 5.

(9) Same as 7, but size restricted as in 4.

Statistical Queries: These queries are useful when conclusions are to be drawn from the data based on statistical facts.

(10) For each pair of displayable entities, D_1 and D_2, involved in a subword conflict (D_1 is the subword of D_2), obtain $p(D_1, D_2)$ = (*number of occurrences of* D_1 *which occur as subwords of* D_2)/(*number of occurrences of* D_1).

(11) For each pair of displayable entities, D_1 and D_2, involved in a prefix-suffix conflict, obtain $q(D_1, D_2)$ = (*number of occurrences of* D_1 *which have prefix-suffix conflicts with* D_2) /(*number of occurrences of* D_1).

If $p(D_1, D_2)$ or $q(D_1, D_2)$ is greater than a statistically determined threshold, then the following could be be said with some confidence: *Presence of* D_1 *implies presence of* D_2.

8 Applications

Circular strings may be used to represent circular genomes [1] such as $G4$ and $\phi X174$. The detection and analysis of patterns in genomes helps to provide insights into the evolution, structure, and function of organisms. [1] analyzes $G4$ and $\phi X174$ by linearizing and then constructing their scdawg. Our work improves upon [1] by :

(i) analyzing circular strings without risking the "loss" of patterns.

(ii) extending the analysis and visualization techniques of [5] for linear strings to circular strings.

Circular strings in the form of chain codes are also used to represent closed curves in computer vision [11]. The objects of Figure 18(a) are represented in chain code as follows:
(1) Arbitrarily choose a pixel through which the curve passes. In the diagram, the starting pixels for the chain code representation of objects 1 and 2 are marked by arrows.
(2) Traverse the curve in the clockwise direction. At each move from one pixel to the next, the direction of the move is recorded according to the convention shown in Figure 18(b).

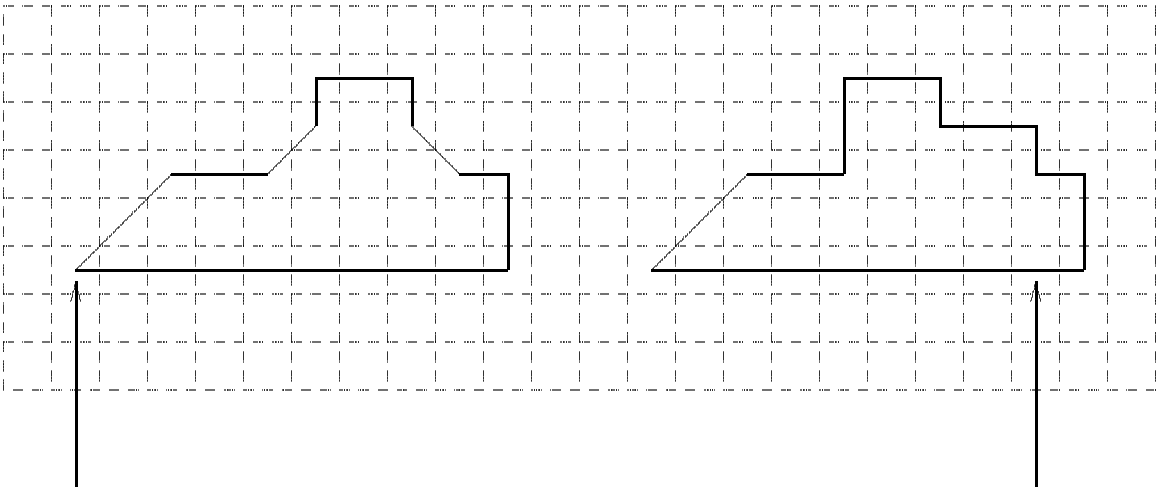
Objects 1 and 2 are represented by *11221022432446666666666* and *66666666611220022422424 46* respectively. The alphabet is $\{0, 1, 2, 3, 4, 5, 6, 7\}$ which is fixed and of constant size (8) and therefore satisfies the condition of Section 2. We may now use the visualization techniques of [5] to compare the two objects. For example, our methods would show that objects 1 and 2 share the segments S1 and S2 (Figure 18(c)) corresponding to *0224* and *2446666666661122* respectively. Information on other common segments would also be available. The techniques of this paper make it possible to detect all patterns irrespective of the starting pixels chosen for the two objects.

Circular strings may also be used to represent polygons in computer graphics and computational geometry [3]. Figure 19 shows a polygon which is represented by the following alternating sequence of lines and angles: $b\beta a\alpha e\alpha e\alpha c\beta c\beta e\beta a\alpha e\alpha e\alpha c\beta c\beta b\alpha c\alpha d\alpha c\alpha$, where α denotes a 90 degree angle and β, a 270 degree angle.

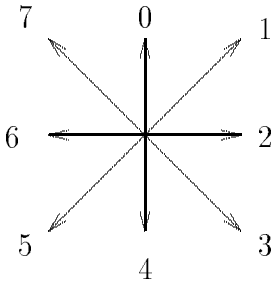
The techniques of this paper would point out all instances of self similarity in the polygon, such as $a\alpha e\alpha e\alpha c\beta c$. Note, however, that for the methods to work efficiently, the number of lines and angles that are used to represent the polygons must be small and fixed.

9 Conclusions

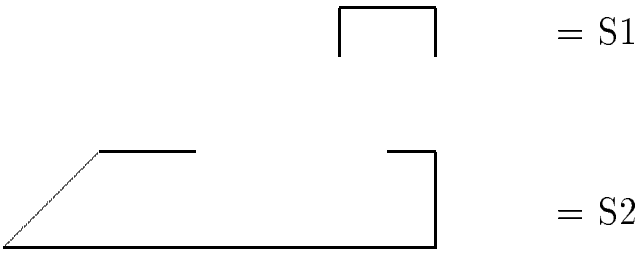
In this paper, we have defined the scdawg for circular strings and shown how it can be used to solve problems in the visualization and analysis of patterns in circular strings. We expect that it can also be used for other string matching applications involving circular strings. An important feature of the scdawg for circular strings is that it is easy to implement and use when corresponding techniques for scdawgs for linear strings are already available.



Starting position for object 1 (a) Starting position for object 2



Chain code representations of directions
(b)



(c)

Figure 18: Representing closed curves by circular strings

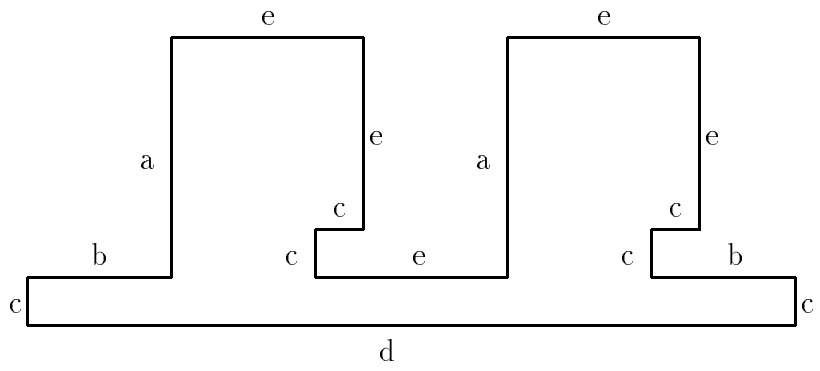


Figure 19: Representing polygons by circular strings

Acknowledgement

We are grateful to Professor Gerhard Ritter for pointing out the application of circular strings to the representation of closed curves.

References

[1] B. Clift, D. Haussler, T.D. Schneider, and G.D. Stormo , "Sequence Landscapes," *Nucleic Acids Research*, vol. 14, no. 1, pp. 141–158, 1986.

[2] G.M. Morris , "The Matching of Protein Sequences using Color Intrasequence Homology Displays," *J. Mol. Graphics*, vol. 6, pp. 135–142, 1988.

[3] S.L. Tanimoto, "A method for detecting structure in polygons," *Pattern Recognition*, vol. 13, no. 6, pp. 389–394, 1981.

[4] D. Mehta and S. Sahni, "String Visualization," *In Preparation*, 1991.

[5] D. Mehta and S. Sahni, "Computing Display Conflicts in String Visualization," *Submitted for journal publication*, 1991.

[6] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht, "Complete Inverted Files for Efficient Text Retrieval and Analysis," *J. ACM*, vol. 34, no. 3, pp. 578–595, 1987.

[7] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas, "The Smallest Automaton Recognizing the Subwords of a Text," *Theoretical Computer Science*, no. 40, pp. 31–55, 1985.

[8] M. E. Majster and A. Reiser, "Efficient on-line construction and correction of position trees," *SIAM Journal on Computing*, vol. 9, pp. 785–807, Nov. 1980.

[9] E. McCreight, "A space-economical suffix tree construction algorithm," *Journal of the ACM*, vol. 23, pp. 262–272, Apr. 1976.

[10] M. T. Chen and Joel Seiferas, "Efficient and elegant subword tree construction," in *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), NATO ASI Series, Vol. F12, pp. 97–107, Berlin Heidelberg: Springer-Verlag, 1985.

[11] R. Gonzalez, P. Wintz, *Digital Image Processing, 2nd Edition.* Addison Wesley, 1987.